Регрессионное тестирование и комбинирование различных уровней тестирования

Регрессионное тестирование

Регрессионное тестирование (Regression Testing) — процесс повторного выполнения ранее проведённых тестов после внесения изменений в ПО (добавление новых функций, исправление ошибок, оптимизация кода, обновление зависимостей), чтобы убедиться, что существующий функционал продолжает работать корректно.

Основные задачи:

- выявить «наведённые» ошибки — дефекты, возникшие из-за изменений в коде;
- подтвердить, что исправление бага не привело к появлению новых проблем;
- проверить стабильность системы после рефакторинга или обновления библиотек/фреймворков;
- обеспечить непрерывное качество продукта на протяжении всего жизненного цикла разработки.

Ключевые стратегии:

1. Полная регрессия — запуск всего набора регрессионных тестов. Подходит для критических релизов, но требует много времени и ресурсов.

2. Выборочная регрессия — выбор тестов, затрагивающих изменённые модули или связанные с ними компоненты. Экономит время, но может пропустить косвенные эффекты.

3. Приоритезация тестов —
сначала выполняются тесты для критически важных функций и высокорисков ых областей.

4. Автоматизация —
ключевой элемент эффективности, особенно в CI/CD-пайплайнах. Автоматиз ированные регрессионные наборы запускаются после каждого коммита или с борки.

5. Анализ влияния изменений —
определение, какие тесты нужно запустить на основе изменённого кода (напр имер, с помощью инструментов покрытия кода).

Инструменты автоматизации:

- Selenium — для веб-UI;
- JUnit/TestNG — для Java-юнит- и интеграционных тестов;
- PyTest — для Python;
- Cypress, Playwright — современные фреймворки для веб-тестирования;
- Jenkins, GitLab CI, GitHub Actions — интеграция в CI/CD.

Комбинирование различных уровней тестирования

Эффективная стратегия тестирования предполагает интеграцию разных у ровней для максимального покрытия дефектов при оптимальном использован ии ресурсов. Уровни дополняют друг друга: каждый нацелен на выявление оп ределённого типа проблем.

Уровни тестирования и их роль:

1. Модульное (юнит-) тестирование:

- Цель: проверка отдельных функций, методов, классов в изоляции.

- Выявляемые дефекты: алгоритмические ошибки, ошибки кодирования, локальные баги.
- Методы: «белый ящик», покрытие кода (statement, branch coverage).
- Роль в регрессии: быстрый фидбек после изменений в модуле; автоматизированные юнит-тесты — основа регрессионного набора.

2. Интеграционное тестирование:

- Цель: проверка взаимодействия между модулями, компонентами, сервисами.
- Выявляемые дефекты: проблемы с интерфейсами, передачей данных, синхронизацией, общими ресурсами.
- Методы: тестирование интерфейсов, API (REST, gRPC), сообщений (MQ).
- Роль в регрессии: проверка, что изменения в одном модуле не нарушили интеграцию с другими.

3. Системное тестирование:

- Цель: проверка всей системы как единого целого в среде, приближённой к продакшн.
- Выявляемые дефекты: отсутствующая функциональность, проблемы производительности, безопасности, совместимости, удобства использования.
- Методы: «чёрный ящик», сквозные сценарии (end-to-end), нагрузочное, стресс-, security-тестирование.
- Роль в регрессии: подтверждение, что система стабильна после изменений; покрытие бизнес-процессов.

4. Приёмочное тестирование (UAT):

- Цель: убедиться, что ПО соответствует бизнес-требованиям и гот ово к релизу.
- Выявляемые дефекты: несоответствие ожиданиям пользователей, пробелы в функциональности.
- Методы: сценарии пользователей, сценарии бизнес-процессов.
- Роль в регрессии: финальная проверка перед выпуском, особенно после крупных изменений.

Принципы комбинирования:

- Пирамида тестирования:
  - основание — много быстрых и дешёвых юнит-тестов (70–80 % набора);
  - середина — меньше интеграционных тестов (15–20 %);
  - вершина — мало системных и приёмочных тестов (5–10 %), но они самые дорогие и медленные.
- Раннее обнаружение дефектов: чем раньше найден баг, тем дешевле его исправить. Юнит-тесты ловят ошибки на этапе написания кода.
- Постепенное расширение охвата: от проверки отдельных модулей к скв озным сценариям.
- Автоматизация критических путей: регрессионный набор должен вклю чать автоматизированные тесты всех уровней, особенно для часто изменяемы х и критически важных частей системы.
- Интеграция в CI/CD:

1. После коммита запускаются юнит-тесты.
2. При успешной сборке — интеграционные тесты.

3. Перед релизом —
системные и приёмочные тесты (частично или полностью).

Пример комбинированного регрессионного тестирования

Сценарий: в веб-приложении «Онлайн-магазин» исправлена ошибка в мод уле расчёта скидки.

Шаги и уровни тестирования:

1. Модульное тестирование:

- запустить юнит-тесты для метода calculateDiscount();
- проверить граничные случаи: нулевая скидка, максимальная скид ка, некорректные входные данные.

2. Интеграционное тестирование:

- проверить взаимодействие модуля скидок с:
- корзиной покупок (CartService);
- модулем оплаты (PaymentService);
- протестировать API-эндпоинт /api/discount с разными запросами.

3. Системное тестирование:

- выполнить сквозной сценарий:
- добавить товары в корзину;
- применить промокод со скидкой;
- оформить заказ;
- проверить итоговую сумму в выписке.
- провести нагрузочное тестирование страницы корзины с примене нием скидок (симуляция 100 пользователей).

4. Регрессионный набор:

- повторно запустить тесты для смежных модулей:
- история заказов (убедиться, что скидка корректно отобража ется);
- профиль пользователя (проверка накопленных бонусов);
- админ-панель (корректность отчётов по скидкам).

5. Приёмочное тестирование (UAT):

- дать пользователям-тестерам выполнить ключевые сценарии со с кидками;
- собрать обратную связь о понятности интерфейса применения пр омокодов.

Результат:

- подтверждено, что исправление ошибки в расчёте скидки не нарушило работу корзины, оплаты, истории заказов и админ-панели;
- проверены не только прямые, но и косвенные эффекты изменения;
- обеспечено соответствие бизнес-требованиям и удобство для пользоват елей.

Вывод: комбинирование уровней тестирования в рамках регрессионной ст ратегии позволяет:

- снизить риск пропуска дефектов;
- оптимизировать затраты на тестирование;
- ускорить цикл разработки за счёт автоматизации и приоритизации;
- поддерживать высокое качество ПО при частых изменениях.